SRM INSTITUTE OF SCIENCE & TECHNOLOGY, NCR CAMPUS

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

LAB RECORD FILE

Name	**:**	**ANANYA GUPTA**
Registration No	**:**	**RA1911003030265**
Section	**:**	**I**
Subject Code	**:**	**18CSC304J**
Title	**:**	**COMPILER DESIGN LAB**
Semester	**:**	**VI**

Academic year 2021 - 2022

SRM
INSTITUTE OF SCIENCE AND TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)
DELHI-NCR CAMPUS, GHAZIABAD (U.P)

SRM INSTITUTE OF SCIENCE & TECHNOLOGY, NCR CAMPUS, MODINAGAR, GHAZIABAD, U.P – 201204

Register No.: **RA1911003030265**

BONAFIDE CERTIFICATE

This is to certify that Lab Report of **Compiler Design Laboratory** (**18CSC304J**), which is submitted by ……ANANYA GUPTA…….. in partial fulfillment of the requirement for the award of degree of B. Tech. in Department of Computer Science & Engineering of SRM Institute of Science & Technology, NCR Campus, Modinagar, is a record of the candidate own work carried out by him/her under our supervision during the academic year 2019-2020.

HEAD OF DEPARTMENT **LAB IN-CHARGE**

*Submitted for the university examination held on*_____.

List of Experiments

Experiment 1: Transition Diagram to Transition Table

Aim: Write a program in C/C++ to show the transition table from a given transition diagram.

Algorithm:

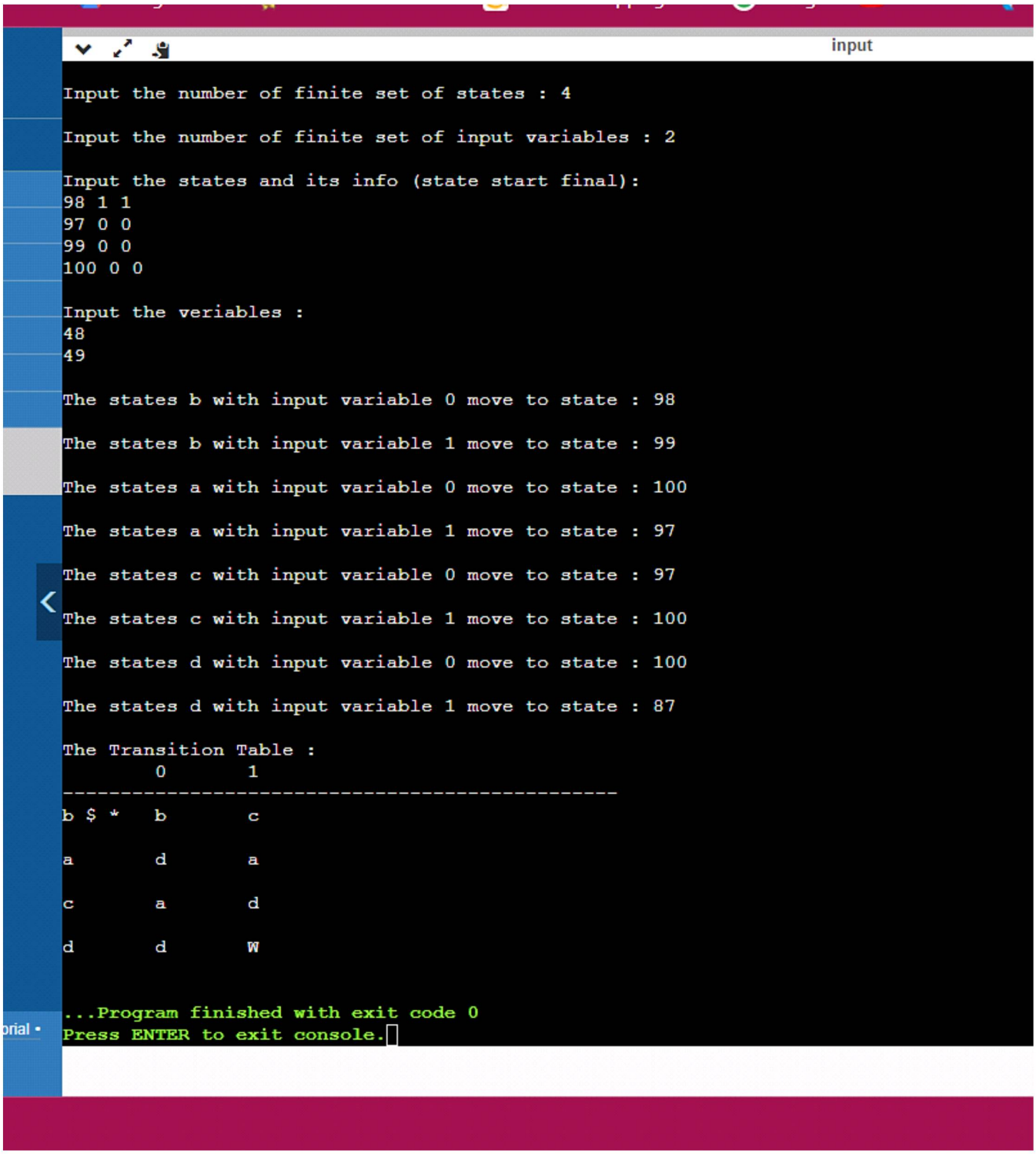
1. Start
2. Enter the number of states.
3. Enter the number of input variables.
4. Enter the state and its information.
5. Enter the input variables.
6. Enter the transition function information i.e., transition value from a state with an input variable.
7. Show the Transition Table.
8. Stop

Code:

```
#include<stdio.h>
#include<stdlib.h>
struct setStates
{
int state;
int final; // 0 - NO 1 - YES
int start; // 0 - NO 1 - YES
};
typedef struct setStates sstate;
void main()
{
int s,v,i,j;
int **sv,*var;
sstate *states;
printf("\nInput the number of finite set of states : ");
scanf("%d",&s);
printf("\nInput the number of finite set of input variables : ");
scanf("%d",&v);
// creating transition table
sv = (int **)malloc(v*sizeof(int));
//printf("\n1 sucess\n");
for(i=0;i<s;i++)
{
sv[i]=(int *)malloc(sizeof(int));
}

states = (sstate *)malloc(s*sizeof(sstate));
printf("\nInput the states and its info (state start final): \n");
for(i=0;i<s;i++)
{
scanf("%d%d%d",&states[i].state,&states[i].start,&states[i].final);
}
// storing input veribale
var = (int *)malloc(v*sizeof(int));
```

```
printf("\nInput the veriables : \n");
for(i=0;i<v;i++)
{
scanf("%d",&var[i]);
}
// storing inputs of transition function
for(i=0;i<s;i++)
{
for(j=0;j<v;j++)
{
printf("\nThe states %c with input variable %c move to state : ",states[i].state,var[j]);
scanf("%d",&sv[i][j]);
}
}
// display transition table on screen
printf("\nThe Transition Table : \n");
printf("\t");
for(i=0;i<v;i++)
{
printf("%c\t",var[i]);
}
printf("\n-----------------------------------------------");
for(i=0;i<s;i++)
{
printf("\n%c %c %c\t",states[i].state,(states[i].start==0)?' ':'$',(states[i].final==0)?' ':'*');
for(j=0;j<v;j++)
{
printf("%c\t",sv[i][j]);
}
printf("\n");
}
}
```

Output:



Result: The Program Executed successfully

Experiment 2: Implementation of Lexical Analyzer

Aim: Write a program in C/C++ to implement a lexical analyzer.

Algorithm:

1. Start
2. Get the input expression from the user.
3. Store the keywords and operators.
4. Perform analysis of the tokens based on the ASCII values.
5.

ASCII Range	TOKEN TYPE
97-122	Keyword else identifier
48-57	Constant else operator
Greater than 12	Symbol

6. Print the token types.
7. Stop

Code:

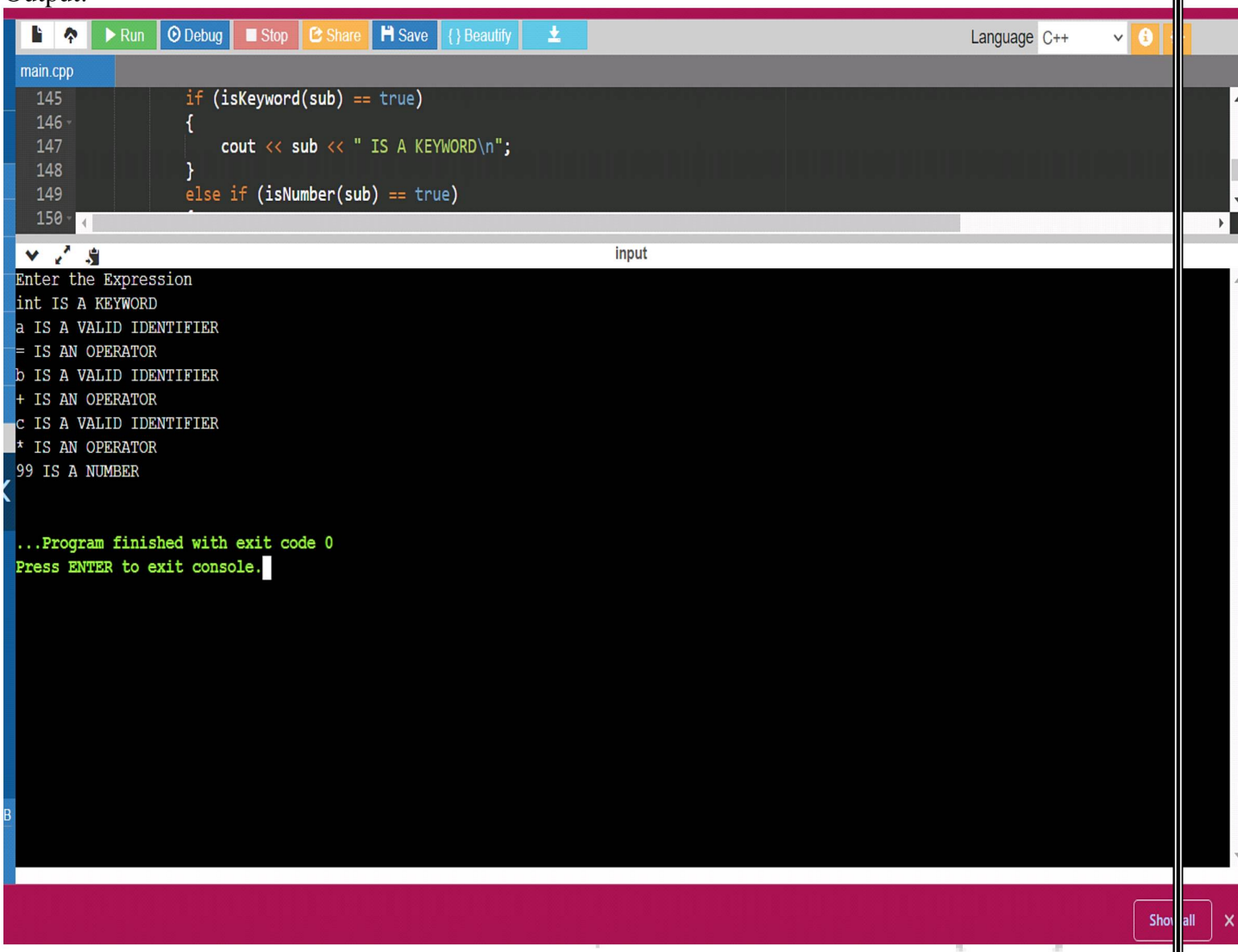
```
#include <fstream>
#include <iostream>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

using namespace std;

bool isPunctuator(char ch) // check if the given character is a punctuator or not
{
   if (ch == ' ' || ch == '+' || ch == '-' || ch == '*' ||
      ch == '/' || ch == ',' || ch == ';' || ch == '>' ||
      ch == '<' || ch == '=' || ch == '(' || ch == ')' ||
      ch == '[' || ch == ']' || ch == '{' || ch == '}' ||
      ch == '&' || ch == '|')
   {
      return true;
   }
   return false;
}

bool validIdentifier(char *str) // check if the given identifier is valid or not
{
   if (str[0] == '0' || str[0] == '1' || str[0] == '2' ||
      str[0] == '3' || str[0] == '4' || str[0] == '5' ||
      str[0] == '6' || str[0] == '7' || str[0] == '8' ||
      str[0] == '9' || isPunctuator(str[0]) == true)
   {
      return false;
   } // if first character of string is a digit or a special character, identifier is not valid
   int i, len = strlen(str);
```

```
    if (len == 1)
    {
        return true;
    } // if length is one, validation is already completed, hence return true
    else
    {
        for (i = 1; i < len; i++) // identifier cannot contain special characters
        {
            if (isPunctuator(str[i]) == true)
            {
                return false;
            }
        }
    }
    return true;
}

bool isOperator(char ch) // check if the given character is an operator or not
{
    if (ch == '+' || ch == '-' || ch == '*' ||
        ch == '/' || ch == '>' || ch == '<' ||
        ch == '=' || ch == '|' || ch == '&')
    {
        return true;
    }
    return false;
}

bool isKeyword(char *str) // check if the given substring is a keyword or not
{
    if (!strcmp(str, "if") || !strcmp(str, "else") ||
        !strcmp(str, "while") || !strcmp(str, "do") ||
        !strcmp(str, "break") || !strcmp(str, "continue") || !strcmp(str, "int") || !strcmp(str, "double") ||
!strcmp(str, "float") || !strcmp(str, "return") || !strcmp(str, "char") || !strcmp(str, "case") || !strcmp(str,
"long") || !strcmp(str, "short") || !strcmp(str, "typedef") || !strcmp(str, "switch") || !strcmp(str,
"unsigned") || !strcmp(str, "void") || !strcmp(str, "static") || !strcmp(str, "struct") || !strcmp(str,
"sizeof") || !strcmp(str, "long") || !strcmp(str, "volatile") || !strcmp(str, "typedef") || !strcmp(str,
"enum") || !strcmp(str, "const") || !strcmp(str, "union") || !strcmp(str, "extern") || !strcmp(str, "bool"))
    {
        return true;
    }
    else
    {
        return false;
    }
}

bool isNumber(char *str) // check if the given substring is a number or not
{
    int i, len = strlen(str), numOfDecimal = 0;
```

```
    if (len == 0)
    {
        return false;
    }
    for (i = 0; i < len; i++)
    {
        if (numOfDecimal > 1 && str[i] == '.')
        {
            return false;
        }
        else if (numOfDecimal <= 1)
        {
            numOfDecimal++;
        }
        if (str[i] != '0' && str[i] != '1' && str[i] != '2' && str[i] != '3' && str[i] != '4' && str[i] != '5'
&& str[i] != '6' && str[i] != '7' && str[i] != '8' && str[i] != '9' || (str[i] == '-' && i > 0))
        {
            return false;
        }
    }
    return true;
}

char *subString(char *realStr, int l, int r) // extract the required substring from the main string
{
    int i;

    char *str = (char *)malloc(sizeof(char) * (r - l + 2));

    for (i = l; i <= r; i++)
    {
        str[i - l] = realStr[i];
        str[r - l + 1] = '\0';
    }
    return str;
}

void parse(char *str) // parse the expression
{
    int left = 0, right = 0;
    int len = strlen(str);
    while (right <= len && left <= right)
    {
        if (isPunctuator(str[right]) == false) // if character is a digit or an alphabet
        {
            right++;
        }

        if (isPunctuator(str[right]) == true && left == right) // if character is a punctuator
        {
```

```
            if (isOperator(str[right]) == true)
            {
                std::cout << str[right] << " IS AN OPERATOR\n";
            }
            right++;
            left = right;
        }
        else if (isPunctuator(str[right]) == true && left != right || (right == len && left != right)) //
check if parsed substring is a keyword or identifier or number
        {
            char *sub = subString(str, left, right - 1); // extract substring

            if (isKeyword(sub) == true)
            {
                cout << sub << " IS A KEYWORD\n";
            }
            else if (isNumber(sub) == true)
            {
                cout << sub << " IS A NUMBER\n";
            }
            else if (validIdentifier(sub) == true && isPunctuator(str[right - 1]) == false)
            {
                cout << sub << " IS A VALID IDENTIFIER\n";
            }
            else if (validIdentifier(sub) == false && isPunctuator(str[right - 1]) == false)
            {
                cout << sub << " IS NOT A VALID IDENTIFIER\n";
            }

            left = right;
        }
    }
    return;
}

int main()
{

    cout << "Enter the Expression" << endl;

    char c[100] = "int a = b + c *99";
    parse(c);
    return 0;
}
```

Output:



Result: The Program Executed successfully

Experiment 3: Computation of FIRST Sets

Aim: Write a program in C/C++ to find the FIRST set for a given set of production rule of a grammar.

Algorithm:
Procedure First

1. Input the number of production N.
2. Input all the production rule *PArray*
3. Repeat steps a, b, c until process all input production rule i.e. *PArray*[N]
    a. If $X_i \neq X_{i+1}$ then
        i. Print Result array of X_i which contain FIRST(X_i)
    b. If first element of X_i of *PArray* is Terminal or ε Then
        i. Add Result = Result U first element
    c. If first element of X_i of *PArray* is Non-Terminal Then
        i. searchFirst(i, *PArray*, N)
4. End Loop
5. If N (last production) then
    a. Print Result array of X_i which contain FIRST(X_i)
6. End

Procedure searchFirst(i, *PArray*, N)

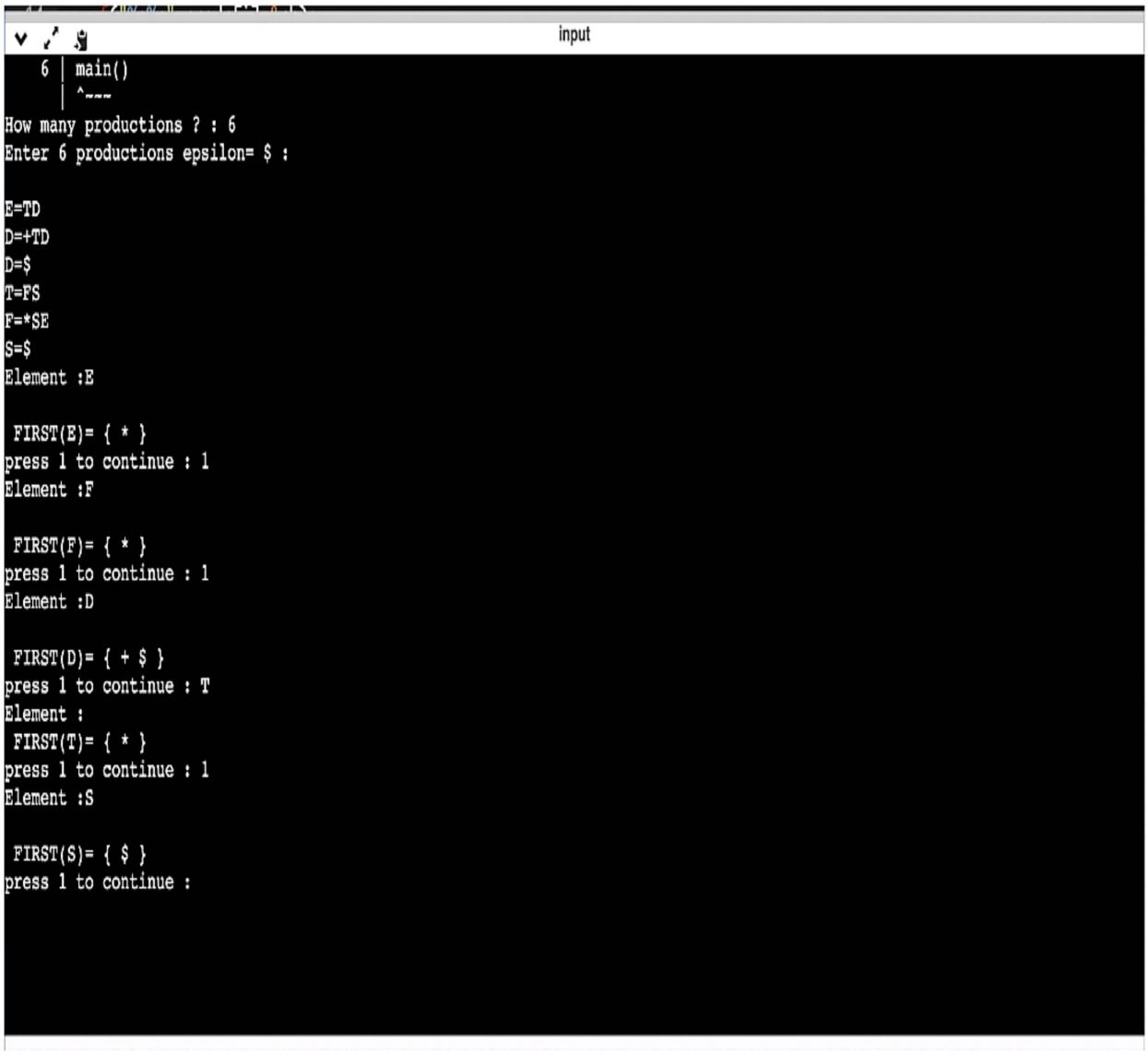
1. Repeat steps Loop j=i+1 to N
    a. If first element of X_j of *PArray* is Non-Terminal Then
        i. searchFirst(j, of *PArray*, N)
    b. If first element of X_j of *PArray* is Terminal or ε Then
        i. Add Result = Result U first element
        ii. Flag=0
2. End Loop
3. If Flag = 0 Then
    a. Print Result array of X_j which contain FIRST(X_j)
4. End

Program:

```
#include<stdio.h>
#include<ctype.h>


void FIRST(char );
int count,n=0;
char prodn[10][10], first[10];

main()
{
int i,choice;
char c,ch;
printf("How many productions ? :");
scanf("%d",&count);
printf("Enter %d productions epsilon= $ :\n\n",count);
for(i=0;i<count;i++)
```

```
scanf("%s%c",prodn[i],&ch);
do
{
n=0;
printf("Element :");
scanf("%c",&c);
FIRST(c);
printf("\n FIRST(%c)= { ",c);
for(i=0;i<n;i++)
printf("%c ",first[i]);
printf("}\n");

printf("press 1 to continue : ");
scanf("%d%c",&choice,&ch);
}
while(choice==1);


}

void FIRST(char c)
{
int j;
if(!(isupper(c)))first[n++]=c;
for(j=0;j<count;j++)
{
if(prodn[j][0]==c)
{
if(prodn[j][2]=='$') first[n++]='$';
else if(islower(prodn[j][2]))first[n++]=prodn[j][2];
else FIRST(prodn[j][2]);
}
}
}
```

Output:



Result: The Program Executed successfully

Experiment 4: Computation of FOLLOW Sets

Aim: Write a program in C/C++ to find a FOLLOW set from a given set of production rule.

Algorithm:

1. Declare the variables.
2. Enter the production rules for the grammar.
3. Calculate the FOLLOW set for each element call the user defined function follow().
4. If x->aBb
    a. If x is start symbol then FOLLOW(x)={$}.
    b. If b is NULL then FOLLOW(B)=FOLLOW(x).
    c. If b is not NULL then FOLLOW(B)=FIRST(b).

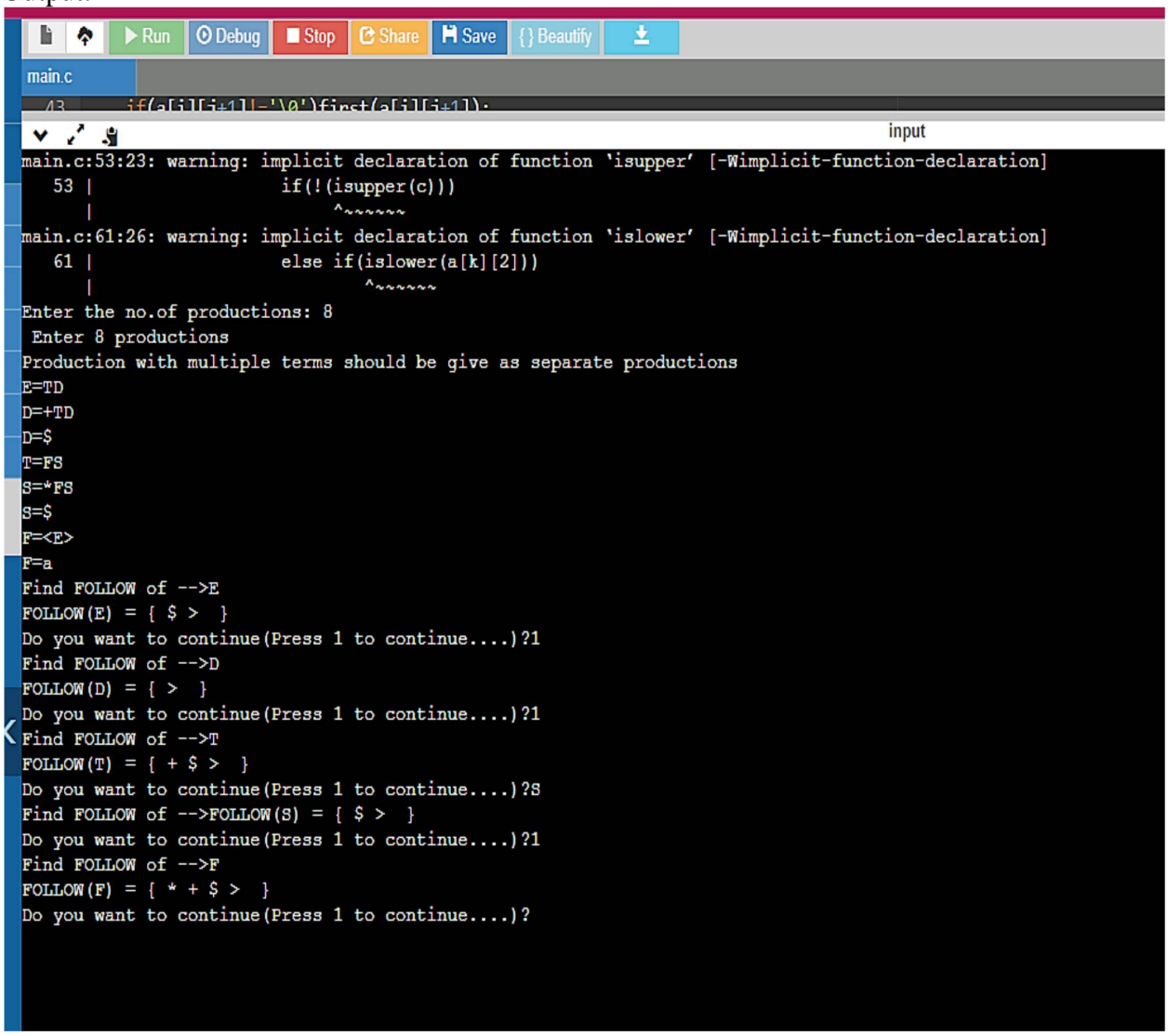
   END.

Program:

```
#include<stdio.h>
#include<string.h>
int n,m=0,p,i=0,j=0;
char a[10][10],followResult[10];
void follow(char c);
void first(char c);
void addToResult(char);
int main()
{
 int i;
 int choice;
 char c,ch;
 printf("Enter the no.of productions: ");
scanf("%d", &n);
 printf(" Enter %d productions\nProduction with multiple terms should be give as separate
productions \n", n);
 for(i=0;i<n;i++)
  scanf("%s%c",a[i],&ch);
    // gets(a[i]);
 do
 {
  m=0;
  printf("Find FOLLOW of -->");
  scanf(" %c",&c);
  follow(c);
  printf("FOLLOW(%c) = { ",c);
  for(i=0;i<m;i++)
   printf("%c ",followResult[i]);
  printf(" }\n");
  printf("Do you want to continue(Press 1 to continue....)?");
 scanf("%d%c",&choice,&ch);
 }
 while(choice==1);
}
void follow(char c)
```

```
{
   if(a[0][0]==c)addToResult('$');
 for(i=0;i<n;i++)
 {
  for(j=2;j<strlen(a[i]);j++)
  {
   if(a[i][j]==c)
   {
    if(a[i][j+1]!='\0')first(a[i][j+1]);
    if(a[i][j+1]=='\0'&&c!=a[i][0])
     follow(a[i][0]);
   }
  }
 }
}
void first(char c)
{
     int k;
              if(!(isupper(c)))
                 //f[m++]=c;
                 addToResult(c);
              for(k=0;k<n;k++)
              {
              if(a[k][0]==c)
              {
              if(a[k][2]=='$') follow(a[i][0]);
              else if(islower(a[k][2]))
                 //f[m++]=a[k][2];
                 addToResult(a[k][2]);
              else first(a[k][2]);
              }
              }
}
void  addToResult(char c)
{
   int i;
   for( i=0;i<=m;i++)
      if(followResult[i]==c)
         return;
  followResult[m++]=c;
}
```

Output:



Result: The Program Executed successfully

Experiment 5: Intermediate Code Generation

Aim: Write a program in C/C++ to generate intermediate code from a given syntax tree statement.

Algorithm:

1. Start the process.
2. Input an expression EXP from user.
3. Process the expression from right hand side to left hand side.
4. FLAG:=0; TOP = -1;
5. IF EXP = '=' then
    i. IF EXP(index – 1) = 0 then
        1. PRINT EXP element from index to (index – 1) and POP STACK[TOP]. Terminate

    Else
    i. PRINT Wrong Expression

    [EndIF]
    IF an operator is found and FLAG = 0 then
    i. TOP:= TOP + 1
    ii. add to STACK[TOP].
    iii. FLAG:=1

    Else
    i. pop twice the STACK and result add to the newID(identifier) and PRINT.
    ii. TOP:=TOP-2. Save newID to STACK[TOP]
    iii. FLAG:=0

    [EndIF]
6. IF an operand is found then
    i. TOP:=TOP+1
    ii. move to STACK [TOP]
    iii. IF TOP > 1 then
        1. pop twice the STACK and result add to the newID(identifier) and PRINT.
        2. TOP:=TOP-2. Save newID to STACK[TOP]
        3. FLAG:=0

    [End]
7. End the process

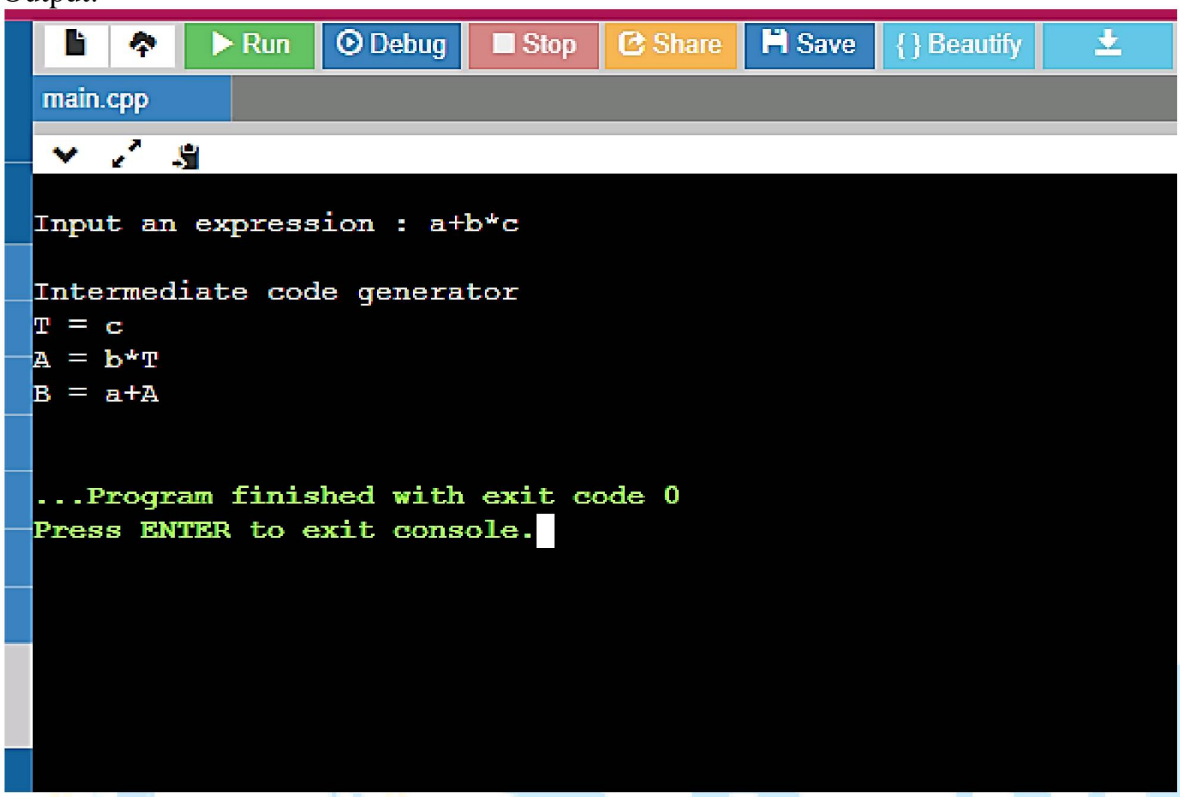
Code:

```
#include<iostream>
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<ctype.h>
using namespace std;
int main()
{
char g,exp[20],stack[20];
int m=0,i,top=-1,flag=0,len,j;
cout<<"\nInput an expression : ";
cin>>exp;
```

```
cout<<"\nIntermediate code generator\n";
len=strlen(exp);
//If expression contain digits
if(isdigit(exp[len-1]))
{
cout<<"T = inttoreal(";
i=len-1;
while(isdigit(exp[i]))
{
i--;
}
for(j=i+1;j<len;j++)
{
cout<<exp[j];
}
cout<<".0)\n";
exp[i+1]='T';
len=i+2;
}
else //If expression having no digit
{
cout<<"T = "<<exp[len-1]<<"\n";
exp[len-1]='T';
}
for(i=len-1;i>=0;i--)
{
if(exp[i]=='=')
{
if((i-1)==0)
{
if(isalpha(stack[top]))
{
cout<<exp[i-1]<<" "<<exp[i]<<" "<<stack[top];
}
else
{
cout<<exp[i-1]<<" "<<exp[i]<<" "<<stack[top]<<stack[top-1];
}
break;
}
else
{
cout<<"\nWrong Expression !";
break;
}
}
if(exp[i]=='+'||exp[i]=='/'||exp[i]=='*'||exp[i]=='-'||exp[i]=='%')
{
if(flag==0)
{
```

```
flag=1;
top=top+1;
stack[top]=exp[i];
}
else
{
g=char('A' + m);
m++;
cout<<g<<" = "<<stack[top]<<stack[top-1]<<"\n";
stack[top-1]=g;
stack[top]=exp[i];
flag=0;
}
}
else
{
top=top+1;
stack[top]=exp[i];
if(top>1)
{
g=char('A' + m);m++;
cout<<g<<" = "<<stack[top]<<stack[top-1]<<stack[top-2]<<"\n";
top=top-2;
stack[top]=g;
flag=0;
}
}
}
return 0;
}
```

Output:



Result: The Program Executed successfully

Experiment 6: Implementation of Shift Reduce Parsing

Aim: Write a program in C/C++ to implement the shift reduce parsing.

Algorithm:

1. Start the Process.
2. Symbols from the input are shifted onto stack until a handle appears on top of the stack.
3. The Symbols that are the handle on top of the stack are then replaces by the left-hand side of the production (reduced).
4. If this result in another handle on top of the stack, then another reduction is done, otherwise we go back to shifting.
5. This combination of shifting input symbols onto the stack and reducing productions when handles appear on the top of the stack continues until all of the input is consumed and the goal symbol is the only thing on the stack - the input is then accepted.
6. If we reach the end of the input and cannot reduce the stack to the goal symbol, the input is rejected.
7. Stop the process.

Program (srp.cpp):

```
#include<stdio.h>
#include<string.h>
int k=0,z=0,i=0,j=0,c=0;
char a[16],ac[20],stk[15],act[10];
void check();
int main()
  {

    puts("GRAMMAR is E->E+E \n E->E*E \n E->(E) \n E->id");
    puts("enter input string ");
    gets(a);
    c=strlen(a);
    strcpy(act,"SHIFT->");
    puts("stack \t input \t action");
    for(k=0,i=0; j<c; k++,i++,j++)
     {
      if(a[j]=='i' && a[j+1]=='d')
        {
          stk[i]=a[j];
          stk[i+1]=a[j+1];
          stk[i+2]='\0';
          a[j]=' ';
          a[j+1]=' ';
          printf("\n$%s\t%s$\t%sid",stk,a,act);
          check();
        }
       else
        {
          stk[i]=a[j];
          stk[i+1]='\0';
          a[j]=' ';
```

```
            printf("\n$%s\t%s$\t%ssymbols",stk,a,act);
            check();
          }
       }

   }
void check()
  {
    strcpy(ac,"REDUCE TO E");
    for(z=0; z<c; z++)
      if(stk[z]=='i' && stk[z+1]=='d')
        {
         stk[z]='E';
         stk[z+1]='\0';
         printf("\n$%s\t%s$\t%s",stk,a,ac);
         j++;
        }
    for(z=0; z<c; z++)
      if(stk[z]=='E' && stk[z+1]=='+' && stk[z+2]=='E')
        {
         stk[z]='E';
         stk[z+1]='\0';
         stk[z+2]='\0';
         printf("\n$%s\t%s$\t%s",stk,a,ac);
         i=i-2;
        }
    for(z=0; z<c; z++)
      if(stk[z]=='E' && stk[z+1]=='*' && stk[z+2]=='E')
        {
         stk[z]='E';
         stk[z+1]='\0';
         stk[z+1]='\0';
         printf("\n$%s\t%s$\t%s",stk,a,ac);
         i=i-2;
        }
    for(z=0; z<c; z++)
      if(stk[z]=='(' && stk[z+1]=='E' && stk[z+2]==')')
        {
         stk[z]='E';
         stk[z+1]='\0';
         stk[z+1]='\0';
         printf("\n$%s\t%s$\t%s",stk,a,ac);
         i=i-2;
        }
   }
```

Output:

```
GRAMMAR is E->E+E
 E->E*E
 E->(E)
 E->id
enter input string
id+id*id-id
stack    input   action

$id        +id*id-id$    SHIFT->id
$E         +id*id-id$    REDUCE TO E
$E+         id*id-id$    SHIFT->symbols
$E+id         *id-id$    SHIFT->id
$E+E          *id-id$    REDUCE TO E
$E            *id-id$    REDUCE TO E
$E*            id-id$    SHIFT->symbols
$E*id            -id$    SHIFT->id
$E*E             -id$    REDUCE TO E
$E               -id$    REDUCE TO E
$E-               id$    SHIFT->symbols
$E-id               $    SHIFT->id
$E-E                $    REDUCE TO E

...Program finished with exit code 0
Press ENTER to exit console.
```

Result: The Program Executed successfully

Experiment 7: Computation of Leading Sets

Aim: Write a program in C/C++ to detect the leading edges of the given set of productions of a grammar.

Algorithm:

1. Start the program.
2. Get the Set of Productions for the grammar from the user. No redundant & cyclic productions must be given.
3. The conditions to be checked are:

Conditions	Inclusions in result
S->Sa	add a
S->Aa	add a, production of A
S->ab	add a
S->AB	Production of A
S->SA	none
S->a	take a
S->SA*	none taken
S->*a	take * leave a

4. Print the Leading edges.
5. Stop the program.

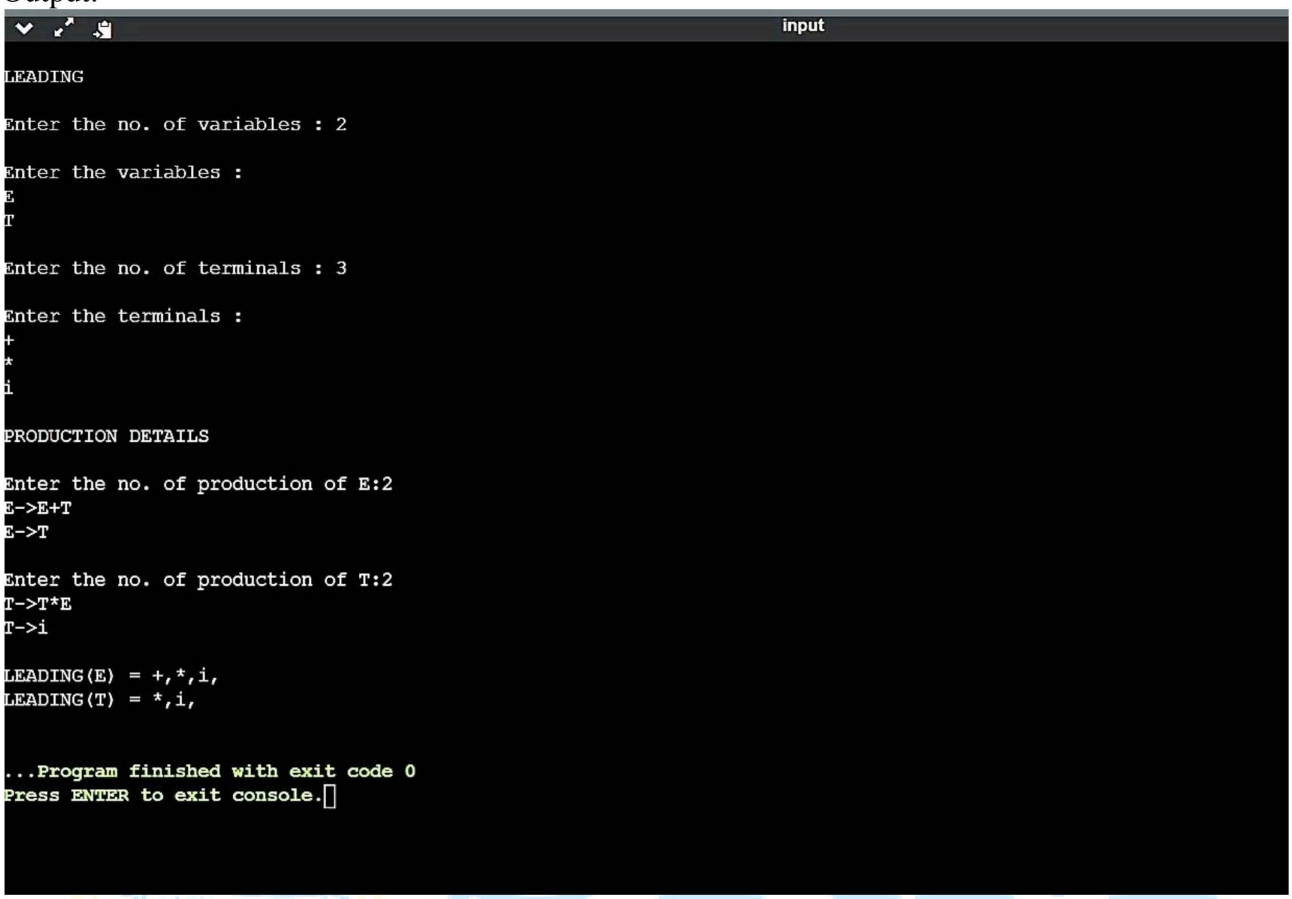
Program (leading.cpp):

```
#include<iostream>
#include<conio.h>
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
using namespace std;

int vars,terms,i,j,k,m,rep,count,temp=-1;
char var[10],term[10],lead[10][10],trail[10][10];
struct grammar
{
int prodno;
char lhs,rhs[20][20];
}
gram[50];
void get()
{
cout<<"\nLEADING\n";
cout<<"\nEnter the no. of variables : ";
cin>>vars;
cout<<"\nEnter the variables : \n";
for(i=0;i<vars;i++)
{
cin>>gram[i].lhs;
var[i]=gram[i].lhs;
}
cout<<"\nEnter the no. of terminals : ";
```

```
cin>>terms;
cout<<"\nEnter the terminals : ";
for(j=0;j<terms;j++)
cin>>term[j];
cout<<"\nPRODUCTION DETAILS\n";
for(i=0;i<vars;i++)
{
cout<<"\nEnter the no. of production of "<<gram[i].lhs<<":";
cin>>gram[i].prodno;
for(j=0;j<gram[i].prodno;j++)
{
cout<<gram[i].lhs<<"->";
cin>>gram[i].rhs[j];
}
}
}
void leading()
{
for(i=0;i<vars;i++)
{
for(j=0;j<gram[i].prodno;j++)
{
for(k=0;k<terms;k++)
{
if(gram[i].rhs[j][0]==term[k])
lead[i][k]=1;
else
{
if(gram[i].rhs[j][1]==term[k])
lead[i][k]=1;
}
}
}
}
for(rep=0;rep<vars;rep++)
{
for(i=0;i<vars;i++)
{
for(j=0;j<gram[i].prodno;j++)
{
for(m=1;m<vars;m++)
{
if(gram[i].rhs[j][0]==var[m])
{
temp=m;
goto out;
}
}
out:
for(k=0;k<terms;k++)
```

```
{
if(lead[temp][k]==1)
lead[i][k]=1;
}
}
}
}
}

void display()
{
for(i=0;i<vars;i++)
{
cout<<"\nLEADING("<<gram[i].lhs<<") = ";
for(j=0;j<terms;j++)
{
if(lead[i][j]==1)
cout<<term[j]<<",";
}
}
cout<<endl;
}
int main()
{

get();
leading();
display();
}
```

Output:



Result: The Program Executed successfully

Experiment 8: Computation of Trailing Sets

Aim: Write a program in C/C++ or Java to detect the trailing edges of the given set of productions of a grammar.

Algorithm:

1. Start the program.
2. Get the Set of Productions for the grammar from the user. No redundant & cyclic productions must be given.
3. Reverse each input productions and print it.
4. The conditions to be checked according to the reversed inputs are:

Conditions	Inclusions in result
S->Sa	add a
S->Aa	add a, production of A
S->ab	add a
S->AB	Production of A
S->SA	none
S->a	take a
S->SA*	none taken
S->*a	take * leave a

5. Print the Trailing edges.
6. Stop the program.

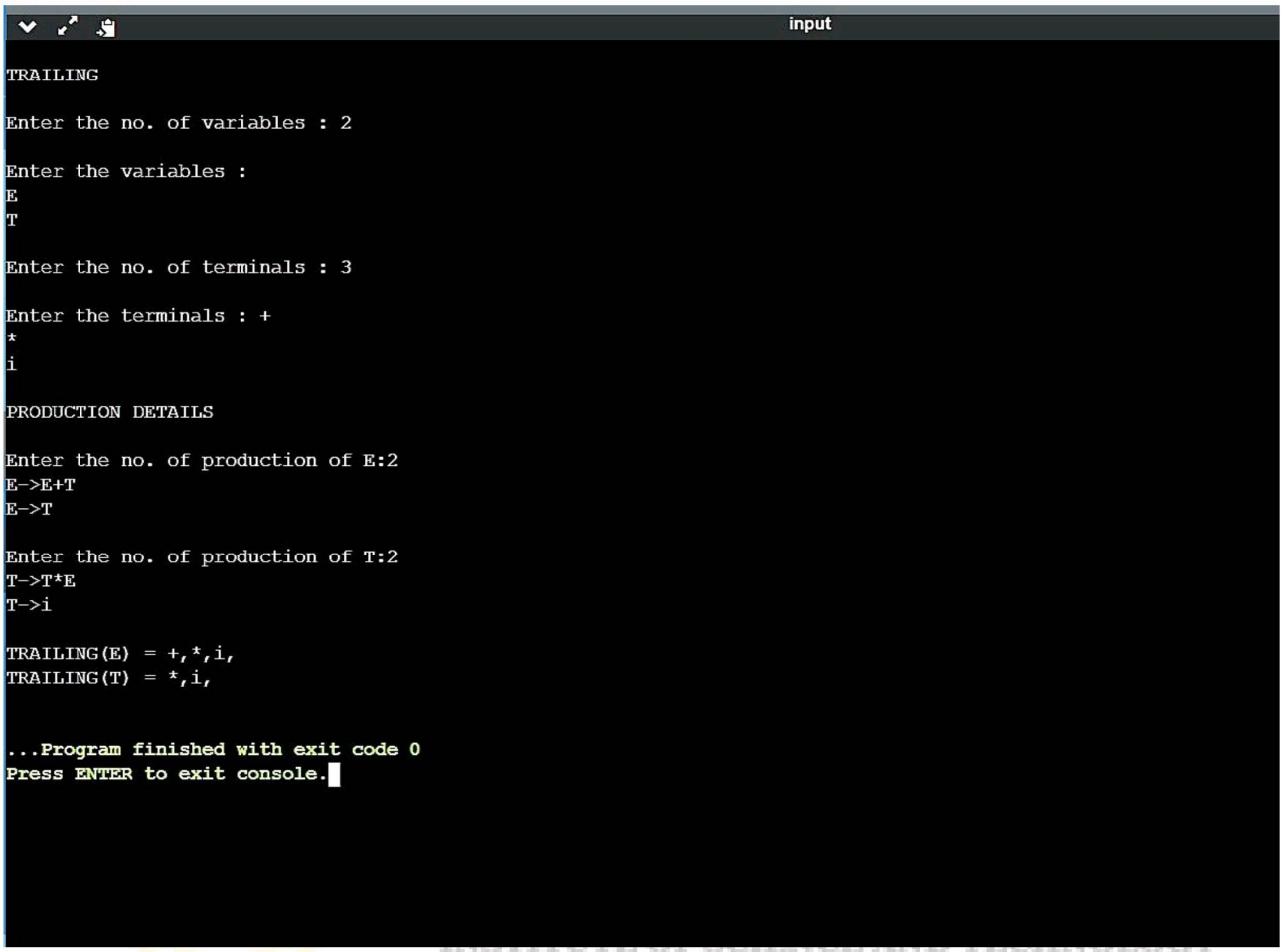
Program (leading.cpp):

```
#include<iostream>
#include<conio.h>
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
using namespace std;

int vars,terms,i,j,k,m,rep,count,temp=-1;
char var[10],term[10],lead[10][10],trail[10][10];
struct grammar
{
int prodno;
char lhs,rhs[20][20];
}
gram[50];
void get()
{
cout<<"\nTRAILING\n";
cout<<"\nEnter the no. of variables : ";
cin>>vars;
cout<<"\nEnter the variables : \n";
for(i=0;i<vars;i++)
{
cin>>gram[i].lhs;
var[i]=gram[i].lhs;
}
```

```
cout<<"\nEnter the no. of terminals : ";
cin>>terms;
cout<<"\nEnter the terminals : ";
for(j=0;j<terms;j++)
cin>>term[j];
cout<<"\nPRODUCTION DETAILS\n";
for(i=0;i<vars;i++)
{
cout<<"\nEnter the no. of production of "<<gram[i].lhs<<":";
cin>>gram[i].prodno;
for(j=0;j<gram[i].prodno;j++)
{
cout<<gram[i].lhs<<"->";
cin>>gram[i].rhs[j];
}
}
}
void trailing()
{
for(i=0;i<vars;i++)
{
for(j=0;j<gram[i].prodno;j++)
{
count=0;
while(gram[i].rhs[j][count]!='\x0')
count++;
for(k=0;k<terms;k++)
{
if(gram[i].rhs[j][count-1]==term[k])
trail[i][k]=1;
else
{
if(gram[i].rhs[j][count-2]==term[k])
trail[i][k]=1;
}
}
}
}
for(rep=0;rep<vars;rep++)
{
for(i=0;i<vars;i++)
{
for(j=0;j<gram[i].prodno;j++)
{
count=0;
while(gram[i].rhs[j][count]!='\x0')
count++;
for(m=1;m<vars;m++)
{
if(gram[i].rhs[j][count-1]==var[m])
```

```
temp=m;
}
for(k=0;k<terms;k++)
{
if(trail[temp][k]==1)
trail[i][k]=1;
}
}
}
}
}
void display()
{
for(i=0;i<vars;i++)
{
cout<<"\nTRAILING("<<gram[i].lhs<<") = ";
for(j=0;j<terms;j++)
{
if(trail[i][j]==1)
cout<<term[j]<<",";
}
}
cout<<endl;
}
int main()
{
get();
trailing();
display();
}
```

OUTPUT:



Result: The Program Executed successfully

Experiment 9: Intermediate code generation - Postfix expression

Aim: Write a program in C/C++ or Java to generate Intermediate Code (Postfix Expression) from given syntax tree.

Code: (in C)

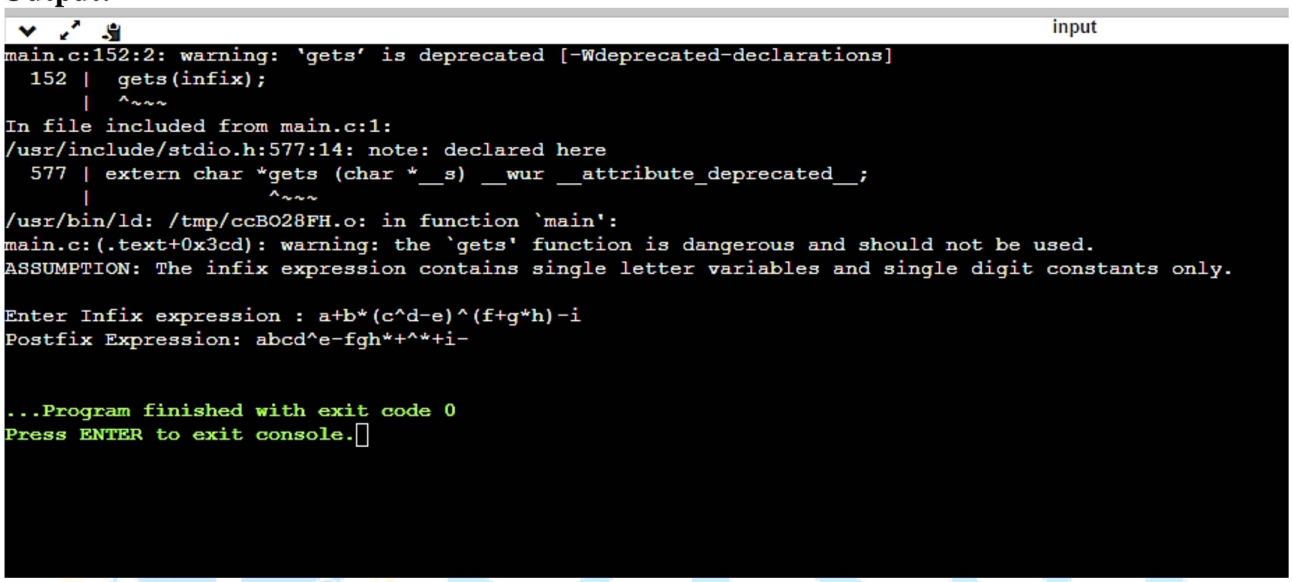
```
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>
#include<string.h>
#define SIZE 100

char stack[SIZE];
int top = -1;

void push(char item)
{
if(top >= SIZE-1)
{
printf("\nStack Overflow.");
}
else
{
top = top+1;
stack[top] = item;
}
}
char pop()
{
char item ;

if(top <0)
{
printf("stack under flow: invalid infix expression");
getchar();
exit(1);
}
else
{
item = stack[top];
top = top-1;
return(item);
}
}
int is_operator(char symbol)
{
if(symbol == '^' || symbol == '*' || symbol == '/' || symbol == '+' || symbol =='-')
{
return 1;
}
```

```
else
{
return 0;
}
}
int precedence(char symbol)
{
if(symbol == '^')
{
return(3);
}
else if(symbol == '*' || symbol == '/')
{
return(2);
}
else if(symbol == '+' || symbol == '-')
{
return(1);
}
else
{
return(0);
}
}

void InfixToPostfix(char infix_exp[], char postfix_exp[])
{
int i, j;
char item;
char x;
push('(');
strcat(infix_exp,")");

i=0;
j=0;
item=infix_exp[i];

while(item != '\0')
{
if(item == '(')
{
push(item)
else if( isdigit(item) || isalpha(item))
{
postfix_exp[j] = item;
j++;
}
else if(is_operator(item) == 1)
{
x=pop();
```

```
while(is_operator(x) == 1 && precedence(x)>= precedence(item))
{
postfix_exp[j] = x;
j++;
x = pop();
}
push(x);

push(item);
}
else if(item == ')')
{
x = pop();
while(x != '(')
{
postfix_exp[j] = x;
j++;
x = pop();
}
}
else
{
printf("\nInvalid infix Expression.\n");
getchar();
exit(1);
}
i++;

item = infix_exp[i];
}
if(top>0)
{
printf("\nInvalid infix Expression.\n");
getchar();
exit(1);
}
if(top>0)
{
printf("\nInvalid infix Expression.\n");
getchar();
exit(1);
}

postfix_exp[j] = '\0';
}
int main()
{
char infix[SIZE], postfix[SIZE];
printf("ASSUMPTION: The infix expression contains single letter variables and single digit constants only.\n");
```

```
printf("\nEnter Infix expression : ");
gets(infix);

InfixToPostfix(infix,postfix);
printf("Postfix Expression: ");
puts(postfix);
return 0;
}
```

Output:



Result: The program executed successfully.

Experiment 10: Implementation of Shift Reduce Parsing

Aim: Write a program in C/C++ to implement the shift reduce parsing.

Algorithm:

1. Start the Process.
2. Symbols from the input are shifted onto stack until a handle appears on top of the stack.
3. The Symbols that are the handle on top of the stack are then replaces by the left-hand side of the production (reduced).
4. If this result in another handle on top of the stack, then another reduction is done, otherwise we go back to shifting.
5. This combination of shifting input symbols onto the stack and reducing productions when handles appear on the top of the stack continues until all of the input is consumed and the goal symbol is the only thing on the stack - the input is then accepted.
6. If we reach the end of the input and cannot reduce the stack to the goal symbol, the input is rejected.
7. Stop the process.

Program (srp.cpp):

```
#include<stdio.h>
#include<string.h>
int k=0,z=0,i=0,j=0,c=0;
char a[16],ac[20],stk[15],act[10];
void check();
int main()
  {

    puts("GRAMMAR is E->E+E \n E->E*E \n E->(E) \n E->id");
    puts("enter input string ");
    gets(a);
    c=strlen(a);
    strcpy(act,"SHIFT->");
    puts("stack \t input \t action");
    for(k=0,i=0; j<c; k++,i++,j++)
     {
      if(a[j]=='i' && a[j+1]=='d')
        {
          stk[i]=a[j];
          stk[i+1]=a[j+1];
          stk[i+2]='\0';
          a[j]=' ';
          a[j+1]=' ';
          printf("\n$%s\t%s$\t%sid",stk,a,act);
          check();
        }
       else
        {
          stk[i]=a[j];
          stk[i+1]='\0';
          a[j]=' ';
```

```
            printf("\n$%s\t%s$\t%ssymbols",stk,a,act);
            check();
          }
       }

   }
void check()
  {
    strcpy(ac,"REDUCE TO E");
    for(z=0; z<c; z++)
      if(stk[z]=='i' && stk[z+1]=='d')
        {
         stk[z]='E';
         stk[z+1]='\0';
         printf("\n$%s\t%s$\t%s",stk,a,ac);
         j++;
        }
    for(z=0; z<c; z++)
      if(stk[z]=='E' && stk[z+1]=='+' && stk[z+2]=='E')
        {
         stk[z]='E';
         stk[z+1]='\0';
         stk[z+2]='\0';
         printf("\n$%s\t%s$\t%s",stk,a,ac);
         i=i-2;
        }
    for(z=0; z<c; z++)
      if(stk[z]=='E' && stk[z+1]=='*' && stk[z+2]=='E')
        {
         stk[z]='E';
         stk[z+1]='\0';
         stk[z+1]='\0';
         printf("\n$%s\t%s$\t%s",stk,a,ac);
         i=i-2;
        }
    for(z=0; z<c; z++)
      if(stk[z]=='(' && stk[z+1]=='E' && stk[z+2]==')')
        {
         stk[z]='E';
         stk[z+1]='\0';
         stk[z+1]='\0';
         printf("\n$%s\t%s$\t%s",stk,a,ac);
         i=i-2;
        }
   }
```

Output:

```
GRAMMAR is E->E+E
 E->E*E
 E->(E)
 E->id
enter input string
id+id*id-id
stack    input   action

$id        +id*id-id$    SHIFT->id
$E         +id*id-id$    REDUCE TO E
$E+         id*id-id$    SHIFT->symbols
$E+id         *id-id$    SHIFT->id
$E+E          *id-id$    REDUCE TO E
$E            *id-id$    REDUCE TO E
$E*            id-id$    SHIFT->symbols
$E*id            -id$    SHIFT->id
$E*E             -id$    REDUCE TO E
$E               -id$    REDUCE TO E
$E-               id$    SHIFT->symbols
$E-id               $    SHIFT->id
$E-E                $    REDUCE TO E

...Program finished with exit code 0
Press ENTER to exit console.
```

Result: The Program Executed successfully

Experiment 11: Construction of DAG

Aim: Write a c or c++ or java to Construct DAG for input expression.

Code:

```
#include<stdio.h>
#include<string.h>
int i=1,j=0,no=0,tmpch=90;
char str[100],left[15],right[15];
void findopr();
void explore();
void fleft(int);
void fright(int);
struct exp
{
 int pos;
 char op;
}k[15];
void main()
{

 printf("\t\tINTERMEDIATE CODE GENERATION OF DAG\n\n");

 scanf("%s",str);
 printf("The intermediate code:\t\tExpression\n");
 findopr();
 explore();

}
void findopr()
{
 for(i=0;str[i]!='\0';i++)
  if(str[i]==':')
  {
  k[j].pos=i;
  k[j++].op=':';
  }
 for(i=0;str[i]!='\0';i++)
  if(str[i]=='/')
  {
  k[j].pos=i;
  k[j++].op='/';
  }
 for(i=0;str[i]!='\0';i++)
  if(str[i]=='*')
  {
  k[j].pos=i;
  k[j++].op='*';
  }
 for(i=0;str[i]!='\0';i++)
```

```
 if(str[i]=='+')
 {
 k[j].pos=i;
 k[j++].op='+';
 }
 for(i=0;str[i]!='\0';i++)
 if(str[i]=='-')
 {
 k[j].pos=i;
 k[j++].op='-';
 }
}
void explore()
{
 i=1;
 while(k[i].op!='\0')
 {
 fleft(k[i].pos);
 fright(k[i].pos);
 str[k[i].pos]=tmpch--;
 printf("\t%c := %s%c%s\t\t",str[k[i].pos],left,k[i].op,right);
 for(j=0;j <strlen(str);j++)
  if(str[j]!='$')
   printf("%c",str[j]);
 printf("\n");
 i++;
 }
 fright(-1);
 if(no==0)
 {
 fleft(strlen(str));
 printf("\t%s := %s",right,left);
 }
 printf("\t%s :=  %c",right,str[k[--i].pos]);

}
void fleft(int x)
{
 int w=0,flag=0;
 x--;
 while(x!= -1 &&str[x]!= '+' &&str[x]!='*'&&str[x]!='='&&str[x]!='\0'&&str[x]!='-
'&&str[x]!='/'&&str[x]!=':')
 {
 if(str[x]!='$'&& flag==0)
 {
 left[w++]=str[x];
 left[w]='\0';
 str[x]='$';
 flag=1;
 }
```

```
 x--;
 }
}
void fright(int x)
{
 int w=0,flag=0;
 x++;
 while(x!= -1 && str[x]!= '+'&&str[x]!='*'&&str[x]!='\0'&&str[x]!='='&&str[x]!=':'&&str[x]!='-
'&&str[x]!='/')
 {
 if(str[x]!='$'&& flag==0)
 {
 right[w++]=str[x];
 right[w]='\0';
 str[x]='$';
 flag=1;
 }
 x++;
 }
}
```

Output:

```
                INTERMEDIATE CODE GENERATION OF DAG

a+b*c/d
The intermediate code:          Expression
        Z := b*c                a+Z/d
        Y := a+Z                Y/d
        Y := d  Y :=  $

...Program finished with exit code 0
Press ENTER to exit console.
```

Result: The Program Executed successfully

Experiment 12: Recursive Descent Parsing

Aim: Write a program in C/ C++ or Java to implement Recursive Descent Parsing

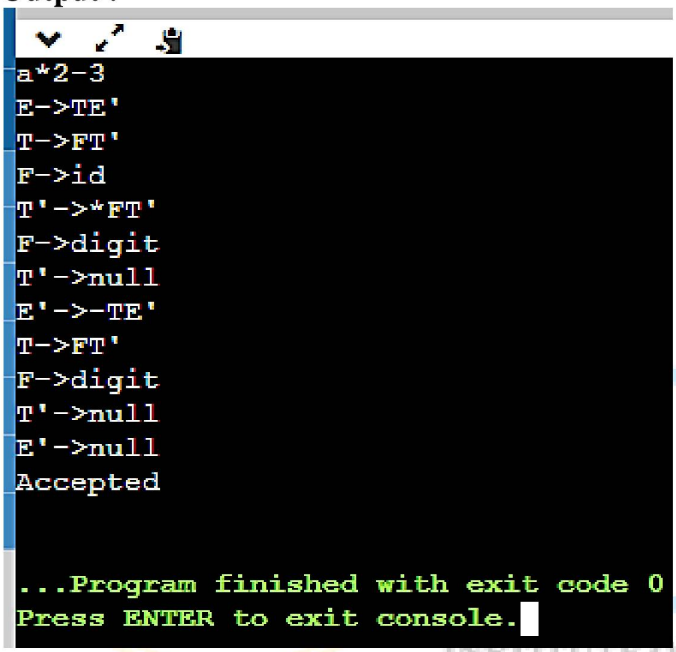
Code :

```
#include <iostream>
#include <stdlib.h>
using namespace std;
/*
E->TE'
E'->+TE'|-TE'|null
T-> FT'
T'->*FT'|/FT'|null
F-> id|num|(E)
*/
int count = 0;
void E();
void Ed();
void T();
void Td();
void F();

string expr;

int main() {
  cin >> expr;
  int l = expr.length();
  expr += "$";
  E();
  if (l == count)
    cout << "Accepted" << endl;
  else
    cout << "Rejected" << endl;
}

void E() {
  cout << "E->TE'" << endl;
  T();
  Ed();
}

void Ed() {
  if (expr[count] == '+') {
    count++;
    cout << "E'->+TE'" << endl;
    T();
    Ed();
  }

  else if (expr[count] == '-') {
```

```
    count++;
    cout << "E'->-TE'" << endl;
    T();
    Ed();
   }

   else {
    cout << "E'->null" << endl;
   }
}

void T() {
  cout << "T->FT'" << endl;
  F();
  Td();
}

void Td() {
  if (expr[count] == '*') {
    count++;
    cout << "T'->*FT'" << endl;
    F();
    Td();
   }

  else if (expr[count] == '/') {
    count++;
    cout << "T'->/FT'" << endl;
    F();
    Td();
   }

  else {
    cout << "T'->null" << endl;
   }
}

void F() {
  if (isalpha(expr[count])) {
    count++;
    cout << "F->id" << endl;
  } else if (isdigit(expr[count])) {
    count++;
    cout << "F->digit" << endl;
  } else if (expr[count] == '(') {
    count++;
    cout << "F->(E)" << endl;
    E();
    if (expr[count] != ')') {
      cout << "Rejected" << endl;
```

```
    exit(0);
   }
   count++;
  } else {
   cout << "Rejected" << endl;
   exit(0);
  }
}
```

Output :



Result: The Program Executed successfully